JUL 15 2004

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	7.Jul.04	MAJOR REPORT

4. TITLE AND SUBTITLE
SERVICE CONTINUITY IN NETWORKED CONTROL USING ETHERWARE

5. FUNDING NUMBERS

6. AUTHOR(S)
CAPT GRAHAM SCOTT R

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
UNIVERSITY OF ILLINOIS AT URBANA

8. PERFORMING ORGANIZATION REPORT NUMBER
CI04-418

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
THE DEPARTMENT OF THE AIR FORCE
AFIT/CIA, BLDG 125
2950 P STREET
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unlimited distribution
In Accordance With AFI 35-205/AFIT Sup 1

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

20040720 077

14. SUBJECT TERMS

15. NUMBER OF PAGES
20

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

THE VIEWS EXPRESSED IN THIS ARTICLE ARE THOSE OF THE AUTHOR AND DO NOT REFLECT THE OFFICIAL POLICY OR POSITION OF THE UNITED STATES AIR FORCE, DEPARTMENT OF DEFENSE, OR THE U.S. GOVERNMENT.

Service Continuity in Networked Control using Etherware

Girish Baliga, Scott Graham, Lui Sha, and P. R. Kumar

University of Illinois at Urbana-Champaign, IL 61801, USA
{gibaliga,srgraham,lrs,prkumar}@uiuc.edu

Abstract. Service continuity is the capability to provide persistent and reliable service, with graceful degradation in the presence of changes. We contend that the implicit need for such a capability is the primary driver of middleware efforts today. This is particularly important for networked control systems interacting with the real world, as they have strict safety requirements. Such systems have to tolerate numerous changes, such as component faults, node failures, and software upgrades, while maintaining operational integrity.
We focus on providing service continuity for networked control systems. The various changes in such systems are classified and illustrated using our traffic control testbed. We then describe how Etherware, our middleware for networked control, handles these changes. Insights into co-design of Etherware, in conjunction with an implementation of our testbed, are presented. The ability of Etherware to provide service continuity, and the associated performance, is demonstrated through illustrative experiments.

Key words: Service continuity, networked control, distributed real-time, component management, fault tolerance, Etherware.

1 Introduction

The ability to tolerate changes is a fundamental quality of a sustainable dynamic system. The useful lifetime of a system is determined to a large extent by its ability to respond to changes in its operating conditions. However, it is impractical to anticipate all such changes before a system is deployed. Instead, systems are often designed to be able to evolve and dynamically adapt to changes in their operating conditions. Software systems are particularly malleable due to the "program as data" concept introduced by the Von Neumann architecture. Indeed, systems software such as operating systems and middleware have been developed to manage application software. This is the basis of most dynamically evolvable software systems today.

Large distributed applications are typically developed as networks of coordinating components using middleware. Component life-cycles are managed using service interfaces provided by the middleware. Components also provide services to each other, and this forms the basis of their interaction. In essence, each component interacts with the rest of the system by providing and consuming services.

However, changes in operating conditions may cause some of these services to be affected. Since components usually depend on the availability of services for proper operation, it becomes necessary to continue to provide some service, even in the presence of changes. Hence, service continuity is a key requirement that needs to be addressed for the sustainability of such applications.

Key innovations in middleware promote service continuity in various ways. Publish-subscribe middleware ([14], [4]) permits servers to be changed without clients having to monitor these changes. Adaptive and reflective middleware ([3], [11]) allows components to modify system configuration, and change their own behavior, while still maintaining service to their clients. Aspect-oriented middleware ([7], [15]) promotes separation of concerns. Aspects addressing a concern such as security, can be added to or removed from component structures, without affecting the services that they provide. Middleware for mobile computing ([13]) addresses continuity of service even during component mobility and system reconfiguration. Thus, much of current research in middleware focuses on maintaining service continuity during change.

Our focus is on networked control systems, which are composed of sensors, actuators, and computers that coordinate over a network, to control a distributed real-time system. Such systems represent a convergence of control with communication and computing, and constitute a logical next step in the evolution of networked systems such as the Internet. Since these systems directly interact with the real world, they are subject to drastic and often unpredictable changes. The systems have to preserve operational integrity in the presence of such changes. The concept of operational integrity captures key non-functional requirements such as safety, availability, and robustness for such systems. To address this, we present Etherware - a middleware for networked control, which we have developed in the IT Convergence Lab at the University of Illinois.

This paper makes three main contributions. First, the notion of operational integrity for networked control is developed, and the main challenges to maintain it are identified. Service continuity is shown to be the key requirement in middleware to support operational integrity for such systems. Second, several middleware issues involved in operational integrity are identified, their influence on the design of Etherware is considered, and its support for service continuity is detailed. A key contribution is the ability to maintain communication channels during component restarts. Finally, a traffic control testbed developed using Etherware is described, and experiments on performance of service continuity for this application are presented.

The presentation is organized as follows. Section 2 defines operational integrity for networked control, and presents key challenges involved. Section 3 addresses the importance of service continuity, middleware design issues, and choices for Etherware. The programming model of Etherware, its architecture, and details about its support for service continuity, are presented in Section 4. A description of our traffic control testbed is provided in Section 5, followed by a presentation of co-design of our application and features in Etherware. Details of experiments to test the support for service continuity in Etherware, are provided

in Section 6. Our contributions are placed in the context of related research in Section 7, and Section 8 concludes.

2 Operational Integrity in Networked Control

This section introduces the notion of operational integrity for networked control systems. The various changes in such systems are classified. The ability to tolerate such changes is presented as a key challenge in maintaining operational integrity for such systems.

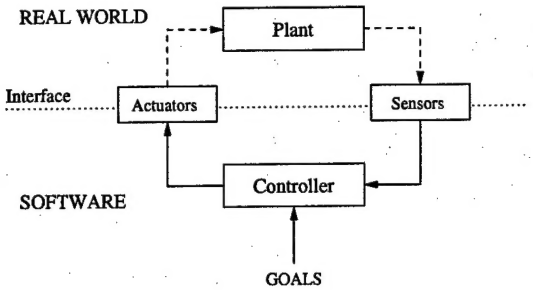


Fig. 1. Schematic of a networked control system

2.1 Operational Integrity

Networked control software interacts with a distributed real-time system, usually called a plant. Sensors provide feedback about the plant behavior. This sensor feedback is used by computers running control programs to generate actuator commands that accomplish desired goals. Actuators implement these commands to control the plant. As shown in Figure 1, sensors and actuators constitute the interface between the software and the real world. Also, the controller is typically implemented as a set of software components operating over a network of computers.

The goals provided to the controller specify the objectives to be achieved during plant operation. An important part of these goals is a set of safety criteria that ensure that the plant operates in a "safe" region. For example, in a traffic control system, a safety criterion would be to maintain a given separation distance between any two cars. This would ensure that there are no collisions between cars. In general, safety criteria depend on specific applications and are usually part of the system specification.

We now define these notions more precisely. The set of variables used to model a plant is called its *state vector*, and the variables are called its *state variables*. The values of state variables of a plant, at a given point in time, represent its

state. The set of values that the state space variables of a plant can take, is called its *state space*. The *safe state space* of a plant, at a given point in time, is the subset of its state space that is specified by the set of safety criteria in its specification. Note that the safe state space may vary with time.

In most networked control systems, however, it is not sufficient to just maintain the system in its safe state space. For example, in the above traffic control system, a safe response to a software failure on one car, would be to stop all cars. This would ensure safety, but it is not the desired response. Hence, a key requirement is to maintain safety without affecting the operation of the system. This can be captured with the following notions. *Operational capability* is the ability of a system to achieve the goals assigned to it. By *operational integrity*, we will mean the property that the state of the plant in a system is always maintained in its safe state space with minimum effect on the operational capability of the system.

2.2 Changes in networked control systems

Networked control systems are subject to various changes during their operation. The changes usually affect one or more components directly. If not handled properly, the changes could further disrupt the operation of other dependent components. Hence, the key challenge in maintaining operational integrity is to minimize the impact of such changes.

The changes occurring in networked control systems can be classified as voluntary or involuntary. *Voluntary* changes are intentionally introduced by a system operator. Component upgrades and configuration changes constitute the principal voluntary changes and may affect one or more of the following:

- **Syntax:** An upgraded component may require syntactic changes such as additional functions or parameters in a service interface. For example, a new controller may require additional information in the updates that it gets from the sensors.
- **Semantics:** Changes in operational semantics may be triggered by changes in operating conditions or upgrades. For example, on detecting a safety violation, the respective controller components may communicate directly to avoid it. Such fault avoidance algorithms may need to be upgraded as the system evolves.
- **Communication:** Components may need to be added to established communication channels at run time. For example, if updates from a sensor are too noisy, then a filter may need to be added to reduce this noise. However, this should not require the controller to re-establish the communication channel to the sensor.

  Components may also want to change quality of service parameters of communication channels. For example, components in wireless networks may want to trade-off reliability for lower delay as shown in [2]. Changes in network topologies could also affect connections between components.

- **Timing:** Changes in operating conditions could cause related changes in timing requirements. For example, the controller may need updates at a higher frequency when approaching critical conditions.
- **Location:** Components may be migrated for better utilization of resources. For example, suppose feedback from a sensor is a lot more frequent than any other communication involving a controller component. The component could then be migrated to the same node as the sensor to reduce network communication traffic.

Involuntary changes are caused by events in the operating environment beyond operator control. These include the following:

- **Passive failures:** These include component crashes due to exceptions, node faults, and failure in communication links.
- **Active failures:** These are usually caused by mis-configured or erroneous components. Another source of active failures is operator mistakes during system reconfiguration.
- **Byzantine failures:** These are caused by malicious components and are usually the hardest changes to cope with.

To maintain operational integrity, the above changes must be handled dynamically, and their impact on the system's operational capability must be minimized.

Notably, active and byzantine failures require semantic information and have to be handled by application specific mechanisms. However, as we demonstrate in the rest of the paper, the other changes can be handled in middleware.

3 Service continuity

This section considers the problem of maintaining operational integrity in networked control. To address this, support for service continuity is presented as the main requirement in middleware. Various issues involved in providing such support are discussed, and their influence on design choices in Etherware are presented.

3.1 Maintaining Operational Integrity

Middleware based systems are typically developed as a set of coordinating components that interact by providing and consuming services. For networked control systems in particular, some of these services may be critical for the operation of a component. For example, the controller in Figure 1 *depends* on getting feedback from the sensors. Controls are usually discrete and calibration is imperfect. Hence, any changes in this feedback service could result in serious faults in the operation of the controller. Hence, continuity of the feedback service is imperative to the operation of the controller. Similarly, the actuators depend on receiving controls from the controller.

Apart from topology reconfiguration, most of the changes listed in Section 2.2 occur at one or more related components. There is little that can be done in middleware to prevent such changes from affecting the respective components. However, the impact of these changes on other dependent components can be minimized. For example, an exception raised in a controller component may cause it to terminate. However, if it is restarted correctly and within a specified time-bound, this fault will not seriously affect the operation of the rest of the system. Such support can be easily provided in middleware. However, the possible support in middleware may be more restricted in pathological cases such as active and byzantine failures. In such situations, typically, application specific mechanisms are also necessary.

In general, apart from acute failures requiring application specific mechanisms, the impact of most changes can be limited to associated components by suitable support from middleware. This mainly consists of maintaining continuity of services consumed and provided by the affected components. Hence, support for service continuity is the primary feature required in middleware for maintaining operational integrity in networked control.

3.2 Design considerations for Service Continuity

Networked control systems have fairly strict safety requirements, and so, components have to respond to changes as soon as possible. For example, on detecting a safety violation, a controller component may not be able to wait for an acknowledgment from another remote component before it decides to take some safe action. On a wireless channel in particular, delays can be fairly large due to interference and fluctuating channel conditions. Hence, to maintain operational integrity, components must be able to operate asynchronously.

Another important consideration is the presence of dependencies in push-based communication channels. For example, the controller cannot wait for the updates from the sensor before sending controls to the actuator. In particular, updates may be delayed or lost due to communication failures in a wireless link. Synchronous communication would require such components to be multi-threaded, or use a fairly complex design involving poller objects for each sensor. Asynchronous operation, on the other hand, eliminates this source of complexity. Based on these considerations, Etherware has been developed as an asynchronous event based middleware.

Event based communication requires a specification of event format. Support for dynamic changes in syntax requires this specification to be flexible and extensible. Service continuity requires this specification to be backward compatible. Based on these requirements, we have used XML [5] as the language for events. All communication in Etherware is through events, which are well-formed XML documents with appropriately defined and extensible formats. For platform independence and due to availability of support for XML, Etherware has been implemented using Java.

Key components may terminate due to voluntary upgrades or involuntary failures. To maintain operational integrity, components need to be restarted and

operational within application specified deadlines. For example, if a controller is restarted, it should know the current state of the plant, as this information may not be entirely available from sensor feedback. Check-pointing is a commonly used technique to address this requirement. Necessary state is periodically check-pointed so that components can be restarted with reasonably current state. To support this, check-pointing has been closely integrated with Etherware design and is provided as a basic service.

Components typically maintain several communication channels with other components. For service continuity, restarting or updating such components should not require these channels to be re-established. Consequently, communication channels should be maintained despite such changes. This is also supported in Etherware. In particular, identifiers for communication channels can be saved as part of check-pointed state. This allows restarted or upgraded components to continue using previously established channels. This also provides communication continuity to other components during these changes.

The necessity to support efficient components restarts has also motivated another basic design choice in Etherware. All components on a given node are managed by a single kernel process with separate threads for components if required. Further, services provided by the middleware also need to be easily restartable and upgradeable. Moreover, invariant aspects of the middleware that cannot be changed dynamically, have to be minimized for maximum flexibility. This has motivated us to adopt a micro-kernel [20] based design for Etherware. This philosophy of flexibility has also resulted in the development of a bare minimum functional interface for components to interact with the middleware. For flexibility and uniformity, all interaction with middleware services is event based.

As we show in the remaining sections, the above design choices have enabled a relatively simple Etherware architecture that provides service continuity by tolerating changes listed in Section 2.2.

4 Etherware

This section describes Etherware, our middleware for networked control. The programming model and the architecture of Etherware are presented. Mechanisms for supporting service continuity in Etherware, in the presence of the changes described in Section 2.2, is detailed.

4.1 Etherware Programming Model

Etherware is an asynchronous event-based middleware. Components communicate by exchanging events which are XML documents, as noted in Section 3.2. Etherware provides a hierarchy of classes that are used to manipulate these documents. The root of this hierarchy is the EtherEvent class that provides various primitives to manipulate the underlying XML document. Application defined events are required to be subclasses of EtherEvent.

Two basic problems attending event delivery in distributed systems are discovery and identification of destination components. The identification problem is solved in Etherware by associating a globally unique id, called a *Binding*, to each component. The discovery problem is solved by associating profiles to addressable components. Each component that needs to be addressed, registers a profile with the middleware.

A profile describes the set of services that a component provides. For example, a profile for a vision sensor could specify that the sensor is a gray-scale camera that covers the region between the points (0,0) and (100,50) in an appropriate coordinate space. Suppose a car controller knows the location of its car to be within coordinates (25,37) and (45,52). It could use this information to connect to a relevant vision sensor using an appropriately defined profile request. Etherware would then match this profile with the profile of the Vision Sensor and forward the connection request to it. Note that the car controller need not specify the type of camera, as this is irrelevant to the service it needs. In general, suitable matching rules have been defined to match profiles.

All EtherEvents have the following three XML tags:

- **Profile:** This identifies the recipient of the event. A profile can be a service description as above or the globally unique Binding of a component.
- **Content:** This represents the contents of the event. All application specific information is contained in this tag.
- **Time-stamp:** Each EtherEvent has a time-stamp associated with it. As an event moves from one node to another, the time-stamp is automatically translated to the local time of the destination.

By default, EtherEvents are delivered reliably and in order. However, there may be event streams that need to be delivered using other specifications. To identify and manipulate a stream of events as a separate entity, Etherware introduces the notion of an *EventPipe*. A component can open an EventPipe to another component and send events through it. EventPipes have settings that can be used to specify how events are delivered through them. For example, the car controller can tolerate a few lost sensor updates for lower delay, but has no use for old updates. Hence, the vision sensor could open an EventPipe to the car controller requesting unreliable in-order delivery. This means that events along this pipe will not be retransmitted if lost, and events arriving late will be discarded.

It may be necessary to modify events in an EventPipe in response to changes in operating conditions. For example, updates from the vision sensor could get noisy due to bad lighting conditions. We should be able to filter out this noise without having to change the sensor or the controller. Etherware supports this by adding *Filters* to EventPipes dynamically. Figure 2 shows the effective configuration after a Kalman filter has been added to the event pipe between the vision sensor and the car controller. Filters can also be added to intercept all events sent to or received by a component.

A component in Etherware is called an *EventHandler*, as it operates by producing and consuming events. The design of a generic EventHandler is shown in

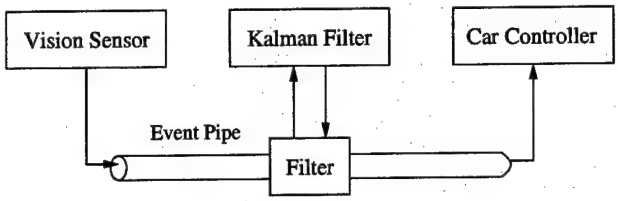


Fig. 2. Filters for EventPipes

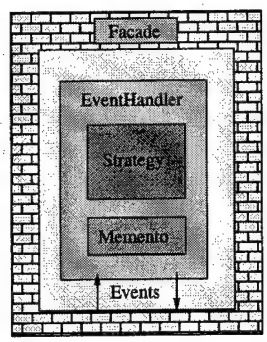


Fig. 3. Design patterns for EventHandlers

Figure 3. This is based on several design patterns [8] that address issues raised in Sections 2.2 and 3.2, where "a design pattern is a solution to a problem in a context" [8]. In software development, several problems may have a common recurring theme. Design patterns represent solutions to such problems that exploit the recurring theme. However, the solutions need to elaborated based on the context of the given problem. We now consider the various problems to be solved in the design of EventHandler and describe how these are addressed using the appropriate design patterns:

- **Memento:** Support for restarts and upgrades requires the ability to capture application state. This is solved by the Memento pattern, wherein component state can be check-pointed and restored on reinitialization.
- **Strategy:** Service continuity requires the ability to replace components without disrupting service. In particular, if the functional interface used to communicate with the component is invariant, then EventHandlers can be replaced dynamically. In this case, the strategy pattern in used in conjunction with the Memento pattern.
- **Facade:** Interaction with various services in the middleware usually requires a component to invoke different sub-systems. This may lead to unnecessary dependencies between the component and middleware sub-systems. This is eliminated by using the Facade pattern to provide a uniform middleware service interface for the components.

EventHandlers can be active or passive. *Passive* EventHandlers do not have any active threads of control. They only respond to incoming events by processing them appropriately and generating resulting events if any.

The interface for a passive EventHandler is as follows:

```
interface PassiveEventHandler {
  /** Initialize the event handler with a given memento */
  public EtherEventList initialize(EtherEvent memento);

  /** Process a given event */
  public EtherEventList processEvent(EtherEvent event);

  /** Terminate process and return a memento if any */
  public EtherEventList terminate();
}
```

The memento of an EventHandler is also defined as a sub-class of EtherEvent. When EventHandlers are instantiated or restarted, the first function called is *initialize*(), whose parameter is its memento. This provides a uniform mechanism to initialize as well as restart components. On receiving an EtherEvent addressed to this EventHandler, the function *processEvent*() is called. For termination, upgrade, or migration, the function *terminate*() is called. All three functions can return one or more EtherEvents addressed to other EventHandlers or the middleware itself. The *terminate*() function, in particular, may return a memento event, representing a check-point for reinitialization.

Active EventHandlers have one or more active threads of control. They can generate events based on activities in their individual threads of control. The interface for an active EventHandler extends $PassiveHandlerInterface$ by including an *activate*() function. This is called to activate additional thread(s) in the EventHandler.

4.2 Etherware Architecture

The architecture of Etherware is based on the micro-kernel concept, as shown in Figure 4. The *Kernel* represents the minimum invariant in Etherware. All other services are implemented as EventHandlers.

The Kernel manages all EventHandlers in a given process. In the current implementation, we have one Etherware process per node. The basic function of the Kernel is to deliver events between its (local) EventHandlers. The Kernel also exposes a service interface, which can be used to manipulate the EventHandlers managed by it. The Kernel has a *Scheduler* that is responsible for scheduling all events and threads. The Scheduler can be replaced dynamically.

As illustrated in Figure 4, each EventHandler is encapsulated in its own Shell. A *Shell* presents a facade to the EventHandler and provides a uniform interface for it to interact with the rest of the system. Shells also encapsulate EventHandler

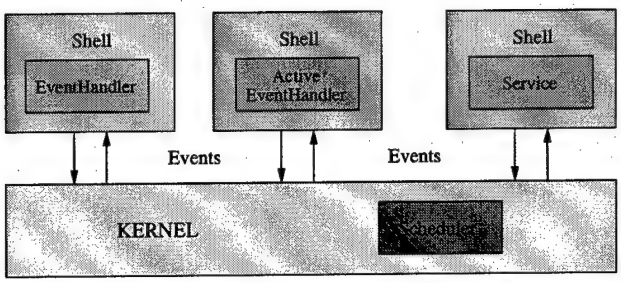


Fig. 4. Architecture of Etherware

specific information such as configurations of EventPipes. Activities involved in component restart, upgrade, and migration are also handled in Shells.

All other functionality in Etherware is provided by service EventHandlers. The following basic services are used during normal operation of Etherware.

- **Profiler:** The Profiler is used to register and look-up profiles of EventHandlers. It is equivalent to a name service.
- **RemoteBus:** The RemoteBus encapsulates all communication with remote nodes over the network. This includes details such as IP addresses, ports, and transport layer protocols. All EtherEvents addressed to remote EventHandlers are forwarded by the Kernel to the RemoteBus. The RemoteBus is an active EventHandler with separate threads to receive events from remote nodes.
- **Timer:** The Timer service is used to translate time-stamps of EtherEvents as they are transmitted from one node to another. To implement this, the Timer is added as a Filter for all events that are sent to and received from the RemoteBus. Time translation is based on computing clock offsets using the Control Time Protocol [9].
- **Ticker:** The Ticker is mostly used by passive EventHandlers for periodic activations. For example, the car controller operates at 10 Hz and has been implemented as a passive EventHandler. For periodic activation, it registers with the Ticker to receive periodic tick events at 100ms intervals. The Ticker can also send one-time alarm events. The Ticker is an active EventHandler.

Since the above services have been implemented as EventHandlers, they can also be restarted or updated dynamically.

4.3 Etherware support for Service Continuity

We now consider how Etherware provides service continuity to support most of the changes listed in Section 2.2.

- **Restarts and upgrades:** In networked control, it is necessary to inform applications about changes such as component upgrades or migration. For

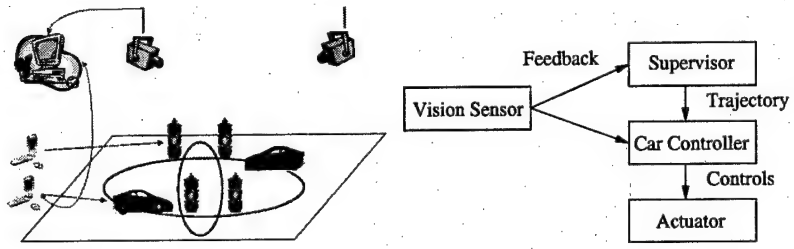
example, if a controller is upgraded, the new controller must know about the current state of the plant. This requires the state of the old controller to be check-pointed before termination. Similarly, migration of a controller component would involve changes in loop delay. This would require changes in parameters such as control gain. Hence, if the controller is migrated without such knowledge, then the plant may become unstable, and thus violate operational integrity of the system. To support this, the EventHandler interface described in Section 4.1 has a *terminate*() function, which is called to check-point state and inform the EventHandler about possible upgrade or migration.

- **Syntax changes:** As discussed in Section 4.1, the functional interface for EventHandlers is simple, uniform, and not expected to change during most operation. Hence, the key source of syntactic change during component upgrade is the formats of events that are consumed and produced by it. However, since event formats are specifications of XML documents, format changes are easily supported. Further, interaction with the underlying XML documents is usually abstracted away by defining appropriate subclasses of EtherEvent. This enables backward compatibility to be addressed using inheritance.
- **Semantic changes:** Service continuity during semantic changes requires a formal specification of application semantics. We are currently working on a formal specification language for Etherware applications.
- **Communication changes:** Changes in communication topologies are supported by Filters and check-point based migration of EventHandlers. Events can be intercepted by defining appropriate Filters. The addition and deletion of Filters does not require the sender or the receiver of the events to be involved. Similarly, migration of EventHandlers does not require communicating components to be informed.
- **Timing changes:** Changes in timing requirements of event delivery are supported by the use of EventPipes. As indicated in Section 4.1, timing behavior of individual threads can be manipulated using the *AbstractScheduler* interface. However, Etherware does not yet support hard real-time deadlines, as our target application operates in soft-real time.
- **Location changes:** Location transparency is supported by globally unique Bindings of EventHandlers.
- **Component failures:** Passive failures such as component exceptions are supported by check-pointing and restart mechanisms in Shells, as noted in Section 4.2. A node failure, on the other hand, triggers appropriate exception events informing remote components connected to EventHandlers on the failed node. These exceptions can be used to handle node failures in application code. For example, all Etherware services described in Section 4.2 are designed to tolerate node restarts.
- **Other failures:** Active and Byzantine failures involve application specific mechanisms and comprehensive support for such changes has not yet been provided in Etherware.

The performance of these mechanisms has been tested through experiments on a prototype traffic control application. The details are presented in following sections.

5 Prototype System

This section presents the prototype traffic control system that has been implemented using Etherware. Issues involved in co-design of Etherware and the application are considered.



(a) Traffic Control Testbed (b) Testbed design

Fig. 5. A Prototype Networked Control System

5.1 Traffic Control Testbed

Figure 5(a) illustrates our traffic control testbed, which is a prototype networked control system. This system consists of a set of remote controlled cars that are driven on a track. Each car is controlled by a dedicated laptop via radio control. A car is driven by controlling its nominal speed and steering angle. Feedback is provided by two ceiling mounted cameras that cover different overlapping portions of the track. The video feed from these cameras is processed by dedicated desktops to track the cars. The computers can communicate using wired or 802.11 wireless networks.

A basic design for the testbed, with three layers of control, is shown in Figure 5(b). Based on desired trajectories of different cars, the supervisor provides goals to the individual car controllers. The goals are sequences of timed locations on the track, called way-points. The individual car controller computes controls to drive the car along these way-points, and sends them to the associated actuator. The actuator controls the car according to the given controls. The vision sensors provide feedback on positions and orientations of cars to the supervisor and the controllers.

A basic safety criterion for this system is to avoid collisions between cars. While the supervisor determines way-points to avoid collisions, numerous changes could still cause collisions to occur. For example, car collisions could occur due to failure of the vision sensor, the controller, or the communication link between them.

We can apply the concepts introduced in Section 2.1 to model this system. The plant of a car controller is the car that it controls. The state variables of the plant are the location, orientation, speed, and steering angle of the car. Similarly, the plant of the supervisor is the set of cars that it supervises. In the testbed, vision sensors monitor locations and orientations of cars. However, a car's speed and steering angle are not monitored. Given this model, operational integrity of the system could require that a minimum distance be maintained between locations of cars, without affecting the ability to control the cars.

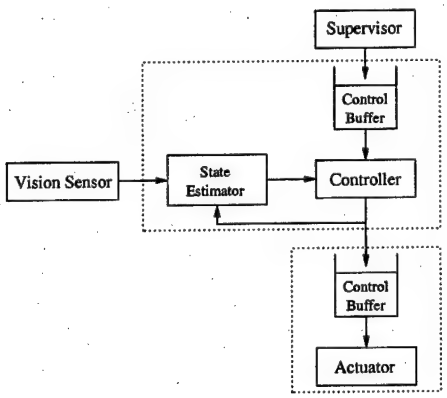


Fig. 6. Modified traffic testbed design

5.2 Issues in Co-design

Networked control systems operate under highly varying conditions. In particular, wireless channels are prone to interference, noise and fading. This leads to unpredictable packet delays and unreliable links. Such problems cannot be entirely addressed in middleware. Rather, maintaining operational integrity during such changes requires application aware mechanisms. For example, in Figure 5(b), a wireless link between the vision sensor and car controller could be lost for a couple of seconds. To maintain operational integrity, the car controller must still be able to control the car using open loop control.

To operate over a wireless network, we need to modify the testbed design of Figure 5(b) to tolerate unpredictable delays and link failures. Figure 6 presents our modified design. The car controller uses a local state estimator to tolerate

noise and jitter in updates from the vision sensor. The state estimator uses a Kalman filter [12] to estimate the current state of the car based on sensor updates and controls sent to the actuator. The estimator can continue to estimate the state of the car even if the link to the vision sensor goes down temporarily. This allows the car controller to tolerate link and component failures in its connection to the vision sensor.

The actuator uses a local control buffer to tolerate changes in the controller. The controller sends a sequence of commands to the actuator every 100 ms. Each sequence has commands for the next two seconds. Hence, the actuator can continue operating the car for two seconds before it needs a new sequence of commands from the controller. However, since the sequence of future commands from the controller is based on imperfect calibration and discrete controls, the safety deadline is usually lesser than two seconds. A similar control buffer isolates the car controller from changes in the supervisor.

The key point is that, the above design is a result of the necessity to maintain operational integrity despite delays, faults, and link failures. These are part of the system operating conditions and must be addressed by application design. Middleware support alone would not be able to provide necessary service continuity without such mechanisms in the application.

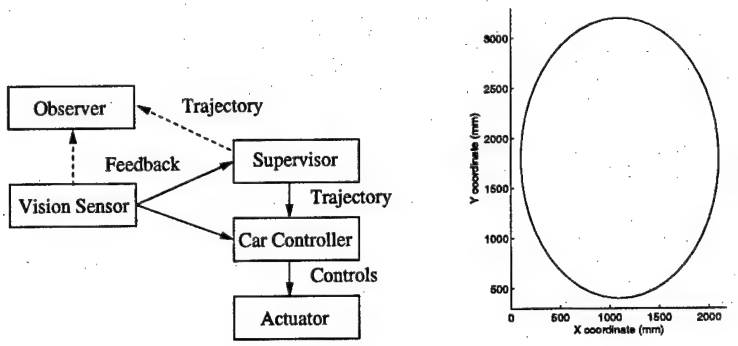
Based on the flexibility afforded by this design, the testbed has been implemented using soft real time control. This also allows Etherware to operate over a general purpose operating system without requiring support for hard deadlines. Consequently, we have focused more on mechanisms, described in Section 4, to support service continuity in the presence of faults and software upgrades. As demonstrated in Section 6, the loss of service due to software restarts or upgrades is minimized by this support, and the involved transients can be easily tolerated by our application.

6 Evaluation

This section presents two experiments evaluating the performance of Etherware mechanisms for service continuity. The first experiment tests an involuntary change involving a component restart, and the second evaluates a voluntary change represented by a component upgrade. Observed results are related to Etherware mechanisms from Section 4.

6.1 Experimental Configuration

From Figure 5(b), we see that the controller is the most constrained component in the traffic testbed.The controller operates under the tightest deadlines and has the highest interconnection complexity in the system. Consequently, changes affecting the controller represent the greatest stress that can be applied to this system. Hence, in the following experiments, we will consider restarting and upgrading the controller while the system is operating. For both experiments, the desired trajectory of the car is the oval shown in Figure 7(b).



(a) Modified testbed design (b) Trajectory

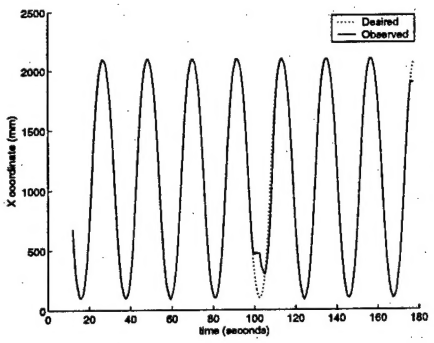
Fig. 7. Experimental setup

Figure 7(a) shows the configuration used to conduct the experiments described in this section. As the controller itself is being affected, we cannot use it to record the experimental trace. Hence, an Observer has been added to the testbed design of Figure 5(b). It monitors the desired trajectory, and the actual position of the car based on vision upgrades, and produces the traces that we analyze. The Observer and the Supervisor were executed on the same node, while all other components were executed on separate computers. All communication was on a dedicated wired network.

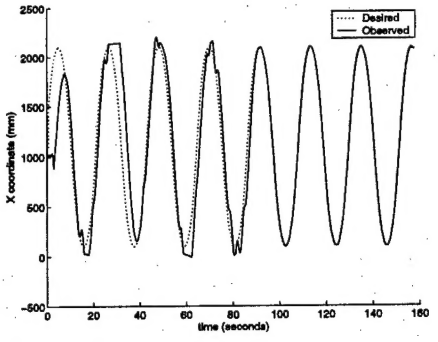
6.2 Controller Restarts

In this experiment, the controller was restarted several times as the car was being driven along the trajectory shown in Figure 7(b). Faults were injected at random by performing an illegal operation (divide by zero) in the Controller. Such a fault caused the Controller to raise an exception and be restarted by Etherware.

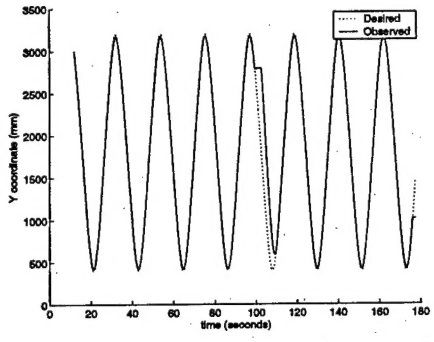
The observations for this experiment are displayed as the first column of plots in Figure 8. Figures 8(a) and 8(c) plot the x and y coordinates of the car position, in millimeters, as a function of time. The dotted lines indicate the desired trajectory. The deviation of the actual car positions from the desired trajectory, as a function of time, is shown in Figure 8(e). The plots are time correlated and a vertical line passing through the three plots identifies the x and y coordinates, as well as the deviation, at a given point in time. Restarts are indicated by the pointers in Figure 8(e), and the accompanying numbers indicate, in milliseconds, the time for restart. These are timestamps at the Observer and include communication and synchronization times between the restarted Controller and the Observer.



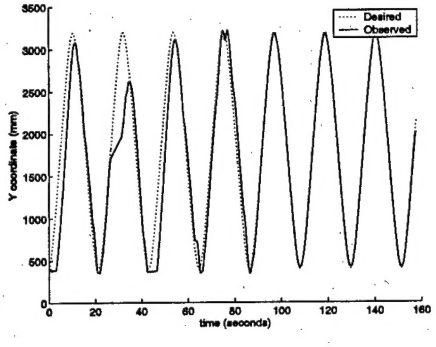
(a) Observations of X-co (restarts)



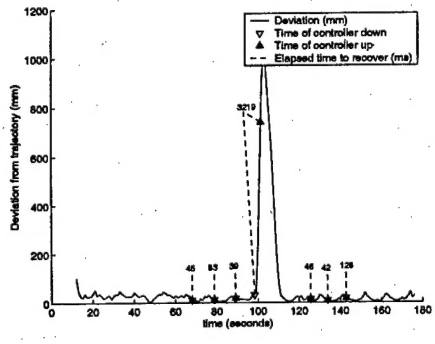
(b) Observations of X-co (upgrade)



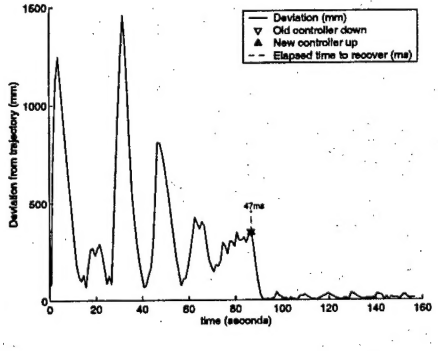
(c) Observations of Y-co (restarts)



(d) Observations of Y-co (upgrade)



(e) Error in trajectory (restarts)



(f) Error in trajectory (upgrade)

Fig. 8. Experimental Results for Controller restarts and upgrade

During the first three iterations of the oval, the Controller was operating normally and we see that the car position tracked the desired trajectory fairly well. The first restart occurred at about 70 seconds into the experiment, and was followed by two other restarts in the next 20 seconds. The last three faults were also handled by the restart mechanisms in Etherware. We see that the error in the car position during these restarts was within the system error bounds during normal operation.

Two of the Etherware mechanisms described in Section 4 contributed to the quick recoveries. First, the Shell intercepted exceptions thrown due to the Controller faults, and restarted it without affecting the EventPipe connections to the other components. Second, before termination, the Controller state was check-pointed according to the Memento pattern, and this check-point was then used for reinitialization.

To illustrate the impact of these two mechanisms, we restarted the Etherware process managing the Controller at about 100 seconds after the start. We see that the restart of Etherware and the Controller took about three seconds, during which the car position accumulated a large error of about 0.8 meters. This illustrates the necessity for efficient restarts. Furthermore, even though the Controller restarted after three seconds, additional error was accumulated before recovery. This was so because, the Controller had to reconnect to the other components, rebuild the state of the car, and bring it back on track. This is the improvement that has been achieved by the check-pointing mechanism.

6.3 Controller Upgrade

In the second experiment, we tested the performance of software upgrade mechanisms in Etherware. The observations for this experiment are shown as the second column of plots in Figure 8. The format of these plots is similar to the plots of Section 6.2.

The car is initially controlled by a coarse Controller that operates myopically. Etherware is then commanded, at about 90 seconds after the start, to upgrade the coarse Controller to a better model predictive Controller. We can easily see the improvement in the car operation. The involved transients are within the system error bounds as well.

This functionality is due to three key Etherware mechanisms from Section 4. First, the Strategy pattern allows one Controller to be replaced by another without any changes to the rest of the system. Second, the Shell is able to upgrade the Controller without affecting the connections to the other components. Finally, the Memento pattern allows the coarse Controller to check-point its state before termination. This is then used to initialize the new Controller. The first mechanism allows for simple upgrades, while the other two mechanisms minimize the impact of the upgrade on other components and the car operation, as shown in Section 6.2.

These experiments clearly demonstrate the need for, and the effectiveness of, Etherware support for service continuity.

7 Related Work

This section presents an overview of related earlier work in this area.

Simplex [19] is an elegant architecture that supports safe dynamic upgrades of control software. Simplex can tolerate timing and semantic faults, and provides run-time error containment using process address space separation. However, component restarts or upgrades still require communication channels to be re-established, and this may affect operational integrity. Support for such functionality could be provided by Etherware as it maintains channels such as EventPipes in the presence of such changes. Designing a class of monitored EventHandlers, based on the Simplex architecture, is an aspect of our current research.

Fault-tolerant CORBA (FT-CORBA) [6] is the primary OMG specification that addresses fault tolerance in distributed systems. The key mechanism in FT-CORBA is to support fault tolerance through redundancy of entities. However, a key problem with this model is that, since replicas execute the same algorithms and have the same inputs, they will have similar failures due to application errors. Hence, safe component restarts is also necessary to support such failures. Some of the other problems that need to be addressed, before FT-CORBA can be used for distributed real-time systems, are considered in [16].

Software frameworks and middleware for networked control in general, are areas of active research. Open Control Platform (OCP) [21] is a Real-Time CORBA [17] based middleware for reconfigurable control systems. Currently OCP is being developed as a software platform for unmanned aerial vehicles (UAVs). While OCP supports service continuity during component and service re-configuration, mechanisms to tolerate faults in application software are not provided. Detailed surveys of related efforts are presented in [18] and [10].

As noted in Section 4.3, support for semantic changes requires a formal specification of application semantics. The development of a formal specification language for Etherware is part of current research. On a related note, the programming model of Etherware is similar to the Actor model of concurrent computation [1]. Like Actors, EventHandlers communicate by sending and receiving events which are buffered in Etherware until consumed. They both have globally unique-ids and modifiable behavior. Based on this, Actors could be used to model and analyze the behavior of Etherware based systems.

8 Conclusions

We have focused on supporting operational integrity in networked control systems. Possible changes occurring in such systems have been classified and illustrated. Service continuity during these changes, has been introduced as a key middleware feature for maintaining operational integrity.

We have presented design considerations in Etherware, our middleware for networked control, and shown how these support service continuity during various changes. The performance of Etherware, during some of these changes, has been evaluated through experiments on a traffic control testbed.

References

1. Agha, G.: Actors - A Model of Concurrent Computation in Distributed Systems. MIT Press, Dec 1986
2. Baliga, G., Graham, S., Sha, L., Kumar, P. R.: Etherware: Domainware for Wireless Control Networks. To appear in Proceedings of ISORC 2004
3. Coulson, G.: What is reflective middleware?. IEEE Distrib. Syst. Online 2, 8, Dec. 2001
4. Cugola, G., Jacobson, H.-Arno.: Using publish/subscribe middleware for mobile systems. ACM SIGMOBILE Mob. Comp. and Comm. Review, 6, 4, Oct 2002, 25–33
5. Extensible Markup Language (XML) 1.0 (Second Edition). W3C - World Wide Web Consortium, Oct 2002
6. Fault Tolerant CORBA Specification, V1.0. Object Management Group (OMG) adopted spec
7. Fleury, M., Reverbel, F.: The JBoss Extensible Server. Middleware 2003
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Paterns - Elements of Reusable Object-Oriented Software. Addison-Wesley Pub Co
9. Time in general-purpose control systems: The Control Time Protocol and an experimental evaluation. Submitted to Proc. of the 43rd IEEE Conf. on Decision and Control, Dec 2004
10. Heck, B., S., Wills, L., M., Vachtsevanos, G., J.: Software Technology for Implementing Reusable, Distributed Control Systems. IEEE Control Systems Magazine, 23, 1, Feb 2003
11. Kon, F., Costa, F., Blair, G., Campbell, R. H.: The case for reflective middleware. Communications of the ACM, v.45 n.6, June 2002
12. Kumar, P., R., Varaiya, P.: Stochastic Systems - Estimation, Identification and Adaptive Control. Prentice-Hall, Inc, 1986.
13. Mascolo, C., Capra, L., Emmerich, W.: Mobile computing middleware. Advanced Lectures on Networking, 20–58
14. Pietzuch, P. R., Shand, B., Bacon J.: A Framework for Event Composition in Distributed Systems. Middelware 2003
15. Popovici, A., Gross, T., Alonso, G.: Dynamic Weaving for Aspect Oriented Programming. 1st Intl. Conf. on Aspect-Oriented Software Development, Apr 2002
16. Gokhale, A., Natarajan, B., Schmidt, D., C., Yajnik, S.: DOORS: Towards High-performance Fault-Tolerant CORBA. Proc. of the 2nd Intl. Sym on Dist. Objects and Appls (DOA '00)
17. Real-Time CORBA Specification Version 2.0. OMG, Inc, Nov 2003
18. Samad, T., Balas, G. (Eds): Software-Enabled Control - Information Technology for Dynamical Systems. IEEE Press, 2003
19. Seto, D., Krogh, B., Sha, L., and Chutinan, A.: Dynamic control system upgrade using the simplex architecture. IEEE Control Systems, 18(4):72-80, 1998
20. Silberschatz, A., Galvin, P. B., Gagne, G.: Operating System Concepts (Sixth Edition). John Wiley & Sons, Inc, June 2001.
21. Wills, L., Sander, S., Kannan, S., Kahn, A., Prasad, J., V., R., Schrage, D.: An Open Control Platform for Reconfigurable, Distributed, Hierarchical Control Systems. Proc. Digital Avionics Systems Conf, Oct 2002.